Homework 1

Due Thursday, 1/31/2019

Before you submit your first solutions, go to `https://submit.cs.pomona.edu/2019sp/cs131` and enroll yourself in the class. When you are ready to turn in your homework, click on the assignment on that page and follow the instructions to submit your work. If you have more than a single file, please put them all in a folder and zip them up before turning them in. On a Mac you can zip up a folder by holding down the control key while pressing the mouse down on the folder. Select Compress to zip up your file. It should be similarly easy on other platforms.

Be sure to test your programs one last time before submitting. I've seen students mess up when commenting code in a way that causes everything to break. Code that doesnt compile will get very little credit. We will be using some automatic testing, and code that doesnt compile brings everything to a halt. With a large class we will not have time to go in to manually tweak your code to make it work.

Your homework solutions should be put in a text file with suffix "hs" (which means it is a Haskell file). Please name your file `hw1MyName.hs` (where MyName is replaced by your name).

Your text file should compile without errors in ghci. This means that in particular, problem numbers and any other non-program text should be given as Haskell comments. Single line comments start with `--`. The comment extends from wherever in the line the `--` occurs to the end of the line. You can also put in block comments. They extend from `{-` to a closing `-}`. For example,

```
silly n = n * n - 3  -- this is a silly comment on a silly function

{-  This is the start of a long
     and important comment
     telling about some code.
-}
```

IMPORTANT: When you write the functions requested below, please make sure that they have the exact names and types specified in the question. If you make an error, your program will crash on our test suite and you will get very little credit. To make extra sure it is written carefully test it on (at least) the example code provided that illustrates what the function does.

Revision to come: I have not included instructions on how to turn in your programs. I will add that before the due date and also post the instructions on Piazza.

1. (10 points) **Haskell types**

   Explain the Haskell type for each of the following declarations:

   (a) a(x,y) = x+2*y
   (b) b(x,y) = x+y/2.0
   (c) c(f) = \y → f y
   (d) d(f,x) = f(f(x))
   (e) e(x,y,b) = if b(y) then x else y

   Because you can simply type these expressions into ghci (with a `let` in front if you type them directly or as is if you load them from a file) to determine their type, be sure to write a short explanation to show that you understand why the function has the type you give.

The answer to this question may be written as a block comment.

2. (50 points) **Haskell Programming**

   For this problem, use the ghci interpreter on the computers in the computer lab (or on your own computer). To run the program in the file "example.hs", type

```
-> ghci
Prelude> :l example.hs
Prelude> :set +t       -- gives more info when you type an expression
```

   at the command line.

   The Haskell compiler will process the program in the file and then wait for the user to type an expression. For example, if "example.hs" contains

```
-- double an integer
double x = x + x;

-- return the length of a list
listLength [] = 0
listLength (l:ls) = 1 + listLength ls
```

   You can test the program by typing the following:

```
*Main> double 10
20
it :: Integer
*Main> listLength (1:[2,3,4])
4
it :: Integer
```

   Start early on this part so you can see the TA or me if you have problems understanding the language. Looking at the examples in the on-line tutorials and text and in your notes will help a great deal in understanding how to use Haskell. Use pattern matching where possible.

   (a) **Basic Functions**

   Define a function `sumSquares` that, given a nonnegative integer n, returns the sum of the squares of the numbers from 1 to n:

```
- sumSquares 4;
30
- sumSquares 5;
55
```

   Define a function `listDup` that takes a pair of an element, e, of any type, and a non-negative number, n, and returns a list with n copies of e:

```
> listDup("moo", 4);
["moo","moo","moo","moo"]
it :: [[Char
> listDup(1, 2);
[1,1]
it :: [Integer]
> listDup(listDup("cow", 2), 2)
[["cow","cow"],["cow","cow"]]
it :: [[[Char]]]
```

Your function will have a type like `(Eq t, Num t) => (a, t) -> [a]`. What does this type mean? Why is it the appropriate type for your function.

(b) **Lists**

Define a recursive function `evens` that returns the list of even numbers that occur in an input list of numbers. For example, `evens [1,2,3,4,8,5,2]` should return [2,4,8,2]. Notice that the elements are returned in the order encountered and duplicates are included. Evens has type

```
evens :: Integral t => [t] -> [t]
```

Define a function `incBy n lst` with type given by `incBy :: Num t => t -> [t] -> [t]`. This function should take parameters n and lst and then increments all of the elements of the list `lst` by n. Thus `incBy 7 [1,4,2]` should return `[8,11,9]`

For extra credit write `incBy'` that behaves exactly as above, but is defined using the built-in `map` function instead of using recursion.

(c) **Zipping and Unzipping**

The function `zip` to compute the pairwise interleaving of two lists of arbitrary length is predefined, but I'd like you to write it from scratch anyway (calling it `zip'`). You should use pattern matching to define this function. The function should have type:

```
zip':: [t] -> [t1] -> [(t, t1)]
```

```
-> *Main> zip' [1,3,5,7] ["a","b","c","de"]
[(1,"a"),(3,"b"),(5,"c"),(7,"de")]
it :: [(Integer, [Char])]
```

Note: If the lists don't have the same length, you may decide how you would like the function to behave. If you don't specify any behavior at all you will get a warning from the compiler that you have not taken care of all possible patterns— this is fine.

Write the inverse function, `unzip'`, which behaves as follows:

```
unzip' :: [(s, t)] -> ([s], [t])

*Main> unzip' [(1,"a"),(3,"b"),(5,"c"),(7,"de")]
([1,3,5,7],["a","b","c","de"])
it :: ([Integer], [[Char]])
```

Again, `unzip` is built-in, but you will write your own `unzip'`.

Write `zip3'`, to zip three lists.

```
zip3' :: [t] -> [t1] -> [t2] -> [(t, t1, t2)]

*Main> zip3' [1,3,5,7] ["a","b","c","de"] [1,2,3,4]
[(1,"a",1),(3,"b",2),(5,"c",3),(7,"de",4)]
it :: [(Integer, [Char], Integer)]
```

Once again, `zip3` is built-in, but you will write your own `zip3'`.

Why can't you write a function `zip_any` that takes a list of any number of lists and zips them into tuples? From the first part of this question it should be pretty clear that for any fixed n, one can write a function `zipn`. The difficulty here is to write a single function that works for all n! I.e., can we write a single function `zip_any` such that `zip_any [list1,list2,...,listk]` returns a list of k-tuples no matter what k is?

(d) **find**

Write a function find that takes a pair of an element and a list and returns the location of the first occurrence of the element in the list (or -1 if it doesn't occur).

```
find :: (Eq a, Eq a1, Num a1) => (a, [a]) -> a1
```

The prefix `(Eq a, Eq a1, Num a1) =>` indicates that the type a of elements of the list must support equality (after all you must check the first argument to see if it is equal to any of the elements of the list). The return type must be some kind of a numeric type as it indicates where in the list the element is found.

For example:

```
*Main> find(3, [1, 2, 3, 4, 5])
2
*Main> find("cow", ["cow", "dog"])
0
*Main> find("rabbit", ["cow", "dog"])
-1
```

First write a definition for find where the element is guaranteed to be in the list. Then, modify your definition so that it returns `-1` if the element is not in the list.